The classes P and NP

Reading

Section 7.2

Practice problems (page 294):

The class P

Computational problems are considered "tractable", if their running time in polynomial. While a polynomial can grow with n, it does so in much more reasonable ways than an exponential.

> The class P consists of all languages that are decidable in polynomial time by a deterministic single-tape Turing Machine.

A number of well-known problems belong to the class P, and looking back at algorithms you have learned in your other classes you can find more examples.

We will now consider several popular members of the class P

The Path problem

The path problem is represented by the language:

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph and has a directed path from } s \text{ to } t\}$$

$$\text{PATH} \in P$$

One relevant question is how we represent the graph G. There are various ways, and they all involve space polynomial in the number of nodes n. Since the class P is effectively invariant under such transformations, we can consider n to be the "size" of our problem.

A "naive" approach to solving this problem would attempt to consider all possible "paths", which are m^m if we denote by m the number of edges. This would not be polynomial (m is essentially $O(n^2)$).

But of course there are more efficient ways, essentially involving marking of the vertices:

1. Start by marking the vertex s.

2. Repeat until nothing new is marked:
    - Go through the edge list.
    - If the source is marked and the target is not, mark the target.
3. See if t is marked.

Clearly the time-intensive portion is the second part. It will have to run once for each vertex (because each time it must mark a new vertex or we are done), and it takes time $O(m)$ to run through the edge list. So it's total running time is $O(n^3)$, polynomial.

Relatively prime numbers

Another popular polynomial-time problem is the determination of whether two numbers are relatively prime or not. An important consideration here is the size of the input.

> A number N is stored in base 2 using $n = O(\log N)$ space, by simply using the base-2 representation of N.

Now consider the problem:

$$\text{RELPRIME} = \{\langle x, y\rangle \mid x, y \text{ are relatively prime } \}$$

$$\text{RELPRIME} \in P$$

The size of the input is here $O(\log N)$ where N is the largest of the two numbers. This is important to keep in mind. For instance a naive approach would be to go through each number d up to x, y and divide into them to see if it is a common factor. But this would take too long: There are in general $O(N)$ such numbers, and $N = 2^{O(n)}$ is exponential in the size of the input. Essentially, it would take too long.

Instead we will perform the well-known Euclidean division algorithm:

> E = On input $\langle x, y\rangle$:
>
> A. Repeat until $y = 0$: 1. Compute $x = x \bmod y$ 2. Swap x and y B. The resulting x is the greatest common divisor of x, y. If it is 1 then we accept, otherwise we reject.

The key intuition here is that each repetition is effectively cutting the size of the inputs x, y by at least a half every second time through the loop. So the number of A steps needed is $O(\log N)$. Each of those steps is also polynomial in $\log N$, the length of the representations of x, y.

Context Free Languages are in P

> If L is a context free language, then it belongs to the complexity class P.

We will only outline the proof here, and refer to the book for details. We start by considering a CFG in Chomsky Normal Form for the language. Then we know that if we want to derive a string of length n, we will require exactly $2n-1$ steps in our derivation.

One naive approach therefore would be to try out all derivations of $2n-1$ steps, but this turns out to not be polynomial in n.

The solution involves the idea of **dynamic programming**, whereupon we store the results of "smaller" problems to avoid having to repeat them. The idea goes as follows:

1. Given a target word $w = w_1 w_2 \cdots w_n$.
2. We will progressively fill up an $n \times n$ table, whose (i, j) entry contains the totality of variables that can derive the substring $w_i w_{i+1} \cdots w_j$. Only one half of the table will need to be filled.
3. The diagonal corresponds to the individual substrings w_i consisting of one character. We can fill those in by a quick scan of the production rules to find any productions $A \to w_i$.
4. For other (i, j) entries: Consider all splits of the substring $w_i w_{i+1} \cdots w_j$, and for each split consider for each rule $A \to BC$ whether B can produce the first part of the split and C can produce the second part. Then A can produce the substring.
5. Repeat this with i, j pairs progressively further from each other (so we fill from the main diagonal and going outwards, one diagonal at a time).
6. w is in L if and only if the start variable S is in the $(1, n)$-th entry.

To examine the complexity of this process, the dominant step is 4. There are in the order of $O(n^2)$ times that step 4 will need to be repeated. It also takes $O(n)$ time to do step 4 (the number of non-terminals in the grammar is constant, and step 4 requires at most that number times n steps). A total running time of $O(n^3)$.

The class NP

The class NP consists of problems that are "verifiable in polynomial time". What this means is that it might not be possible to determine in polynomial time, given a string, whether it is an instance of the language, but it might be possible to verify that if we are provided some more "evidence".

One of the most important such problems is the *Hamiltonian Path* problem HAMPATH, consisting of all representations of triples $\langle G, s, t \rangle$ where G is a directed graph that contains a Hamiltonian path from s to t. A Hamiltonian path is one that passes through all vertices exactly once.

Noone knows a polynomial-time way to determine if a triple as above is in HAMPATH or not. However, it would be possible, if someone provided us with such a path, to

verify that it is indeed a Hamiltonian path. In that sense, this problem is verifiable in polynomial time.

Another problem, called COMPOSITE, consists of all integers that are the product of two smaller integers. It is much easier to verify that a number is composite if you are given the two factors, as opposed to having to find them first.

These ideas lead us to the following formal definition:

> A **verifier** for a language A is an algorithm V such that:
>
> $$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$
>
> We measure the time of the verifier in terms of the length of the string w. A **polynomial time verifier** runs in polynomial time in the length of w. A language is called **polynomially verifiable** if it has a polynomial time verifier.
>
> The string c is often called a **certificate** or **proof** of membership in A, or sometimes a witness.

> NP is the class of all languages that have polynomial time verifiers.
>
> The term stands for **nondeterministic polynomial time**, because there is an alternative characterization using non-deterministic polynomial time Turing machines. The certificate effectively amounts to the non-deterministic machine taking some deterministic steps first to write the correct certificate on the tape.

> A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing machine.

We leave the details to the reader. They are outlined at page 266 of the book.

It should be clear that every language in P is also in NP. Whether they are equal is at this point unknown, and considered one of the hardest questions in theoretical computer science:

> It is not known whether $P = NP$ or whether $P \neq NP$.

Some NP problems

We already considered two NP problems earlier, HAMPATH and COMPOSITE. We will now look at some more:

$$\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ a graph with a } k - \text{clique}\}$$

A k-clique is a set of points that are all adjacent to each other.

It is easy to see that this problem is in NP: All we need as a certificate is the set of k vertices that form a clique, and then it is easy to verify that there is the right number of them and that they have all required edges.

SUBSET-SUM